A Pragmatic Type System for Deductive Verification

Jean-Christophe Filliâtre[1,2], Léon Gondelman[1⋆], Andrei Paskevich[1,2]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] Inria Saclay – Île-de-France, Orsay, F-91893

Abstract. In the context of deductive verification, it is customary today to handle programs with pointers using either separation logic, dynamic frames, or explicit memory models. Yet we can observe that in numerous programs, a large amount of code fits within the scope of Hoare logic, provided we can statically control aliasing. When this is the case, the code correctness can be reduced to simpler verification conditions which do not require any explicit memory model. This makes verification conditions more amenable both to automated theorem proving and to manual inspection and debugging.
In this paper, we devise a method of such static aliasing control for a programming language featuring nested data structures with mutable components. Our solution is based on a type system with singleton regions and effects, which we prove to be sound.

1 Introduction

In this paper, we explore how far we can go with the simplicity behind Hoare logic [1]. This simplicity, which is not just the simplicity of the rules, but foremost, of the proof obligations that stem whereof, is embodied in the rule for assignment:

$$\{\, P[x \leftarrow E] \,\}\ x := E\ \{\, P \,\}$$

Here, we presume that the memory location referred to by x has no other name in P. Once we abandon this hypothesis, that is, when we allow aliases, this simplicity is lost. Over the years, numerous approaches to deductive verification in presence of aliases have been proposed, including explicit memory models [2], separation logic [3], or dynamic frames [4].

However, we can observe that a vast majority of code we may consider verifying still fits in Hoare logic. The secret is abstraction. A structure implementing a mutable set may use arbitrary pointers (say, an AVL tree or a hash table). Yet client code using a mutable set need not be aware of this complexity: it manipulates the set using abstract functions as if it were a single mutable variable, in the sense of Hoare logic. Consequently, we can expect at least some parts of the program to be verified using simple techniques *à la* Hoare logic. How large can

⋆ This work is partially supported by the Bware project (ANR-12-INSE-0010, `http://bware.lri.fr/`) of the French national research organization (ANR).

this part be? It is not realistic to require it to be completely alias-free. However, we can still adapt and adopt the assignment rule above, provided we know statically all aliases for variable x. In contrast with the above-mentioned approaches, which embed the frame conditions into proof obligations, we want to perform a static control of aliases prior to generation of verification conditions. In this way, we regain the simplicity of Hoare logic.

In this paper, we develop such a static control of aliases for a programming language featuring nested data structures with mutable components. Our solution is based on a type system with effects and singleton regions, so that the identity of a mutable value is stored in its type, rather than in its name. In practice, effects and regions can be inferred automatically, thus hiding the added complexity of the typing rules from the programmer. This is how this type system is implemented in the verification tool Why3 [5], where user-written type annotations do not mention regions at all.

Let us illustrate our approach on a small example. Consider a hash table h, implemented as a structure in which one of the fields, named *data*, is an array containing the hash table entries. Assigning the array $h.data$ to a new variable a

var $a = h.data$

gives to a the same type as $h.data$, accounting for the fact they both refer to the same array. Using this information, a Hoare-style verification condition generator will know to update both h and a for any subsequent modification of either $h.data$ or a. Let us see what happens if we change the alias structure by assigning to $h.data$ a different array:

$h.data \leftarrow \textsc{createArray}(10)$

One possible solution consists in changing the type of $h.data$, and thus h, in the rest of the computation. This is known as *strong update* [6]. However, this approach requires dependent types once we start handling assignments under conditions. In the following code snippet

var $a = h.data$
if $\textsc{isFull}(h)$ **then begin**
  var $b = \textsc{createArray}(2 \times \textsc{length}(a))$
  ...transfer the table entries from a to b...
  $h.data \leftarrow b$
end

array $h.data$ is dissociated from a if and only if the condition is true, which we cannot know statically.

Instead of making strong updates, we opt for a different solution. We detect potential *aliasing conflicts* between two names—either two names of the same type become unaliased, or two names of different types become aliased—and we prohibit the further use of one of these names in the rest of the computation. The assignment $h.data \leftarrow b$ in the code fragment above contains two aliasing conflicts. First, a and $h.data$ (which have the same type and thus share the same region) are no longer aliased. Second, b and $h.data$ inhabit distinct regions (according to their types) but are now aliased. To ensure consistency, our type system makes it illegal to mention both a and b anywhere in the code executed after the assignment. However, we can still refer to $h.data$, which now does not have to change its type, since there is no other name to claim it.

Note that we could have invalidated $h.data$, and thus h, instead and preserved a and b. However, since the aliasing conflicts came as the result of a modification of h, we presume that the programmer's intention is to keep h.

Technically, the invalidation is expressed as a *reset effect* of the assignment $h.data \leftarrow b$. Assuming ρ_1 is the region of both a and $h.data$, ρ_2 is the region of b, and ρ is the region of h, the type system associates to the assignment the effect $(\mathit{writes}\ \{\rho\} \cdot \mathit{reset}\ \{\rho_1, \rho_2\})$. This effect makes it illegal to use in the subsequent code any existing variable from which ρ_1 or ρ_2 are reachable without passing through ρ. In this way, a and b are invalidated whereas h is not.

Interestingly enough, the freshness of a newly allocated region ρ can be expressed in our type system with the effect $(\mathit{writes}\ \varnothing \cdot \mathit{reset}\ \{\rho\})$. Indeed, this forbids all existing names that refer to ρ, so that no aliasing conflicts can arise.

Our approach does not apply to arbitrary pointer-based data structures. As we track mutable values through their types, we require that the type of any value includes the regions of all its individual mutable components. In particular, we do not consider recursive mutable data types, such as linked lists or trees. As explained above, we rely on abstraction barriers to provide suitable interfaces to such data structures, so that the remaining code can be type-checked and verified in our system.

The rest of the paper is organized as follows. Section 2 introduces a small language with nested regions and gives a formal description of its semantics and type system. Section 3 states and proves the correctness theorem for this type system. We overview the related work in Section 4 and conclude in Section 5. Proofs of lemmas are given in the appendix.

2 A Small Language with Regions

In this section, we give a formal presentation of our approach. We introduce a small programming language featuring nested data structures with mutable components. We present its syntax and semantics, and we define a type system with regions and effects that formalizes the ideas presented above.

2.1 Syntax

The syntax of the language is given in Fig. 1. Expressions are either atomic or compound terms like conditional, local binding (which subsumes sequence), dynamic allocation, function call, and parallel assignment. The latter allows us to simultaneously modify several fields of several records. The syntax follows a variant of *A-normal* form [7]: In compound expressions, except local binding and conditional branches, all sub-terms must be atomic. This does not reduce expressiveness, since expressions such as $(p(42)).f$ can be rewritten as

$$
\begin{array}{rll}
e ::= & a & \text{atomic expression} \\
| & a.f & \text{field access} \\
| & a.\{f \leftarrow a, \dots, f \leftarrow a\}, \dots, a.\{f \leftarrow a, \dots, f \leftarrow a\} & \text{parallel assignment} \\
| & \{f = a, \dots, f = a\} & \text{record allocation} \\
| & \texttt{let}\ x = e\ \texttt{in}\ e & \text{local binding} \\
| & \texttt{if}\ a\ \texttt{then}\ e\ \texttt{else}\ e & \text{conditional} \\
| & p(a, \dots, a) & \text{function application}
\end{array}
$$

$$
\begin{array}{rll}
a ::= & x & \text{variable} \\
| & v & \text{value}
\end{array}
\qquad
\begin{array}{rll}
v ::= & \ell & \text{store location} \\
| & c & \text{scalar constant}
\end{array}
\qquad
\begin{array}{rll}
c ::= & \mathbb{Z} & \text{integer} \\
| & \texttt{True}, \texttt{False} & \text{Boolean} \\
| & \texttt{()} & \text{unit}
\end{array}
$$

Fig. 1. Syntax.

$\texttt{let}\ x = p(42)\ \texttt{in}\ x.f$. For the sake of readability, we often relax the A-normal form in examples. For instance, the following expression allocates two fresh records, respectively bound to variables x and y, and then swaps the contents of the fields $x.f$ and $y.g$.

$$\texttt{let}\ x = \{\, f = 1 \,\}\ \texttt{in let}\ y = \{\, g = 2 \,\}\ \texttt{in}\ x.\{\, f \leftarrow y.g \,\},\ y.\{\, g \leftarrow x.f \,\}$$

For a given expression e, we denote the sets of free variables, function names, and store locations in e with $\mathcal{F}_v(e)$, $\mathcal{F}_p(e)$, and $\mathcal{F}_\ell(e)$, respectively. Only variables can be bound in expressions. We call e *closed* when $\mathcal{F}_v(e)$ is empty.

2.2 Semantics

We equip our language with a small-step operational semantics, given in Fig. 2. It defines a relation $\mu \cdot e \longrightarrow \mu' \cdot e'$ where μ, μ' are memory stores and e, e' are closed expressions. A *memory store* μ is a partial map that, given a location ℓ and a field f, returns a value, written $\mu(\ell.f)$.

We presume to have a fixed set P of global functions. Primitive functions, such as arithmetic operations or comparisons, operate on scalar values and do not modify the store. An application of a primitive function q is evaluated using a predefined interpretation of q, denoted $[\![q]\!]$ (rule E-Op). Each non-primitive function p is given a definition, denoted $p(x_1, \dots, x_n) \mapsto e$, where we require $\mathcal{F}_v(e) \subseteq \{x_1, \dots, x_n\}$, $\mathcal{F}_\ell(e) = \varnothing$, and $\mathcal{F}_p(e) \subseteq P$. Calls to defined functions are evaluated by definition expansion (rule E-δ).

Rules for allocation and assignment impose that the field names are pairwise distinct within each record. This prevents ambiguity in the resulting store. The semantics allows us to share field names among records, so that it is fine to allocate $\{f = 1; g = 2\}$ and $\{f = 1; h = 3\}$.

We call a sequence (possibly empty) of field names a *path*. Paths are denoted with letter π. An empty path is denoted ϵ. We write $\pi_1 \preceq \pi_2$ to denote that π_1 is a prefix (not necessarily proper) of π_2. We generalize the store access function to paths as follows:

$$\mu(\ell.\pi) \triangleq \begin{cases} \ell & \text{if } \pi = \epsilon \text{ and } \ell \in \operatorname{dom} \mu \\ \mu(\ell'.\pi') & \text{if } \pi = f\pi' \text{ and } \mu(\ell.f) = \ell' \end{cases}$$

$$\frac{}{\mu \cdot \texttt{if True then } e_1 \texttt{ else } e_2 \longrightarrow \mu \cdot e_1}(\text{E-T}) \qquad \frac{\mu(\ell.f) = v}{\mu \cdot \ell.f \longrightarrow \mu \cdot v}(\text{E-Field})$$

$$\frac{}{\mu \cdot \texttt{if False then } e_1 \texttt{ else } e_2 \longrightarrow \mu \cdot e_2}(\text{E-F}) \qquad \frac{[\![q]\!](c_1, \ldots, c_n) = c}{\mu \cdot q(c_1, \ldots, c_n) \longrightarrow \mu \cdot c}(\text{E-Op})$$

$$\frac{}{\mu \cdot \texttt{let } x = v \texttt{ in } e \longrightarrow \mu \cdot e[x/v]}(\text{E-}\zeta) \qquad \frac{p(x_1, \ldots, x_n) \mapsto e}{\mu \cdot p(v_1, \ldots, v_n) \longrightarrow \mu \cdot e[x_i/v_i]}(\text{E-}\delta)$$

$$\frac{\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'}{\mu \cdot \texttt{let } x = e_1 \texttt{ in } e_2 \longrightarrow \mu' \cdot \texttt{let } x = e_1' \texttt{ in } e_2}(\text{E-Ctx})$$

$$\frac{\ell \notin \operatorname{dom} \mu \qquad f_i \text{ are pairwise distinct}}{\mu \cdot \texttt{\{}f_i = v_i\ {}^{i \in [1,\ldots,n]}\texttt{\}} \longrightarrow \mu[\ell.f_i \mapsto v_i] \cdot \ell}(\text{E-Alloc})$$

$$\frac{\ell_i.f_{i,j} \in \operatorname{dom} \mu \qquad \ell_i \text{ are pairwise distinct} \qquad \forall i.\ f_{i,j} \text{ are pairwise distinct}}{\mu \cdot \ell_i.\texttt{\{}f_{i,j} \leftarrow v_{i,j}\ {}^{j \in [1,\ldots,k_i]}\texttt{\}}^{i \in [1,\ldots,n]} \longrightarrow \mu[\ell_i.f_{i,j} \mapsto v_{i,j}] \cdot \texttt{()}}(\text{E-Assign})$$

Fig. 2. Semantics.

A location ℓ' is said to be *accessible* from ℓ in μ when there exists a path π such that $\mu(\ell.\pi) = \ell'$. Given a set of locations L, we denote the set of locations accessible from locations in L with $\mathcal{A}_\ell(\mu \cdot L)$. By abuse of notation, we write $\mathcal{A}_\ell(\mu \cdot e)$ for $\mathcal{A}_\ell(\mu \cdot \mathcal{F}_\ell(e))$.

As is standard in operational semantics, evaluation does not necessarily terminate on a value. Besides non-termination due to recursive functions, there also exist irreducible expressions such as $\texttt{\{}f = 1\texttt{\}}.g$. Our type system will later rule out such irreducible expressions.

2.3 Type System

The purpose of a type system is to ensure that "well-typed programs cannot go wrong" (Milner, 1978). In addition to the standard soundness property, we want our type system to distinguish individual mutable values and to ensure that this static alias identification is preserved by evaluation of well-typed terms. To this end, we introduce a type system with effects, where the typing judgment is

$$\Gamma \cdot \Sigma \vdash e : \tau \cdot \varepsilon$$

Here, expression e is assigned a type τ and an effect ε with respect to a *variable typing environment* Γ (a total mapping from variables to types) and a *store typing environment* Σ (a total mapping from locations to regions). For convenience, we extend store typing to scalar constants: For any c, $\Sigma(c)$ stands for the type of c. This provides us with a uniform notation for the type of store contents.

Types are defined in Fig. 3. Constant values are assigned scalar types: integer, Boolean, and unit type. Store locations are assigned structured data types which we call *regions*. A region consists of a set of fields $f_1, \ldots, f_n$, each field f_i being assigned a type τ_i. Every region carries a unique identifier r. This identifier does

$\tau ::= \nu$	scalar type	$\nu ::= \texttt{Int} \mid \texttt{Bool} \mid \texttt{Unit}$	scalar types
$\mid\ \rho$	region	$\rho ::= \texttt{\{}f : \tau, \ldots, f : \tau\texttt{\}}_r$	record types

Fig. 3. Types and regions.

not have any special meaning and only serves to distinguish types of distinct store locations. In other words, regions are singleton types. The intention behind this is to provide a one-to-one correspondence between regions and memory locations used inside a program.

Given a record type $\rho = \texttt{\{}\ldots, f : \tau, \ldots\texttt{\}}_r$, we write $\rho.f$ to denote τ. If ρ does not contain field f, $\rho.f$ is undefined. Similarly, $\nu.f$ is undefined for any scalar type ν. We extend this notation to paths: $\tau.\epsilon \triangleq \tau$ and $\tau.f\pi \triangleq (\tau.f).\pi$. We write $\mathcal{R}(\tau)$ to denote the set of all regions occurring in τ.

We say that two types τ_1 and τ_2 are *structurally equal* when they are equal up to region identifiers, and we write then $\tau_1 \simeq \tau_2$. Equivalently, two types τ_1 and τ_2 are structurally equal when for any path π, $\tau_1.\pi$ is defined if and only if $\tau_2.\pi$ is defined, and if $\tau_1.\pi$ or $\tau_2.\pi$ is a scalar type then $\tau_1.\pi = \tau_2.\pi$.

A *region substitution* θ is a finite injective map between structurally equal types such that for every region $\rho \in \operatorname{dom}\theta$ and every π, either $\theta(\rho.\pi) = \theta(\rho).\pi$ or both $\rho.\pi$ and $\theta(\rho).\pi$ are undefined.

Along with its type, every expression carries an effect ε, defined as a pair $(\omega \cdot \varphi)$, where ω is the set of regions possibly modified in e (*write effect*) and φ is the set of regions whose use is restricted in the subsequent computation (*reset effect*). We require ω and φ to be disjoint. When the expression e is pure, both sets are empty, and we write $\varepsilon = \bot$.

Every function p is provided with a *type signature* $\tau_1 \times \ldots \times \tau_n \to \tau \cdot (\omega \cdot \varphi)$, where $\tau_1, \ldots, \tau_n$ are the types of formal parameters, τ is the type of the result, and $(\omega \cdot \varphi)$ is the latent effect of the function. If p is a primitive function, then $\tau_1, \ldots, \tau_n, \tau$ must all be scalar types, and both ω and φ must be empty. If p is a defined function, we require that $\omega \subseteq \mathcal{R}(\tau_1, \ldots, \tau_n)$, $\varphi \subseteq \mathcal{R}(\tau_1, \ldots, \tau_n, \tau)$, and $\mathcal{R}(\tau) \setminus \mathcal{R}(\tau_1, \ldots, \tau_n) \subseteq \varphi$. The former two conditions limit the latent effect to the exposed regions, and the last condition requires every fresh region in the result to be reset.

We also require any function definition $p\texttt{(}x_1, \ldots, x_n\texttt{)} \mapsto e$ to be consistent with the signature of p, namely that $\Gamma[x_i \mapsto \tau_i\ {}^{i \in \{1,\ldots,n\}}] \cdot \Sigma \vdash e : \tau \cdot (\omega \cdot \varphi \cup \varphi'')$, where φ'' is the additional reset effect which accounts for the regions introduced in e and not exposed in the type signature, i.e., $\varphi'' \cap \mathcal{R}(\tau_1, \ldots, \tau_n, \tau) = \varnothing$. The invariant on effects (disjointness of write and reset effects) and the properties of the effect union ensure that any writes into these regions disappear from the effect of e. Since locations cannot occur in function definitions and any region in a function body either comes from a parameter or is allocated locally (possibly via a function call) and thus is reset, this justifies the condition $\omega \subseteq \mathcal{R}(\tau_1, \ldots, \tau_n)$.

The typing rules are given in Fig. 4. The rule T-Let for $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ ensures that regions in e_2 are valid with respect to the effects of e_1, according to the following definition:

$$\frac{\Gamma(x) = \tau}{\Gamma \cdot \Sigma \vdash x : \tau \cdot \bot}\,(\text{T-Var}) \qquad \frac{\Sigma(c) = \nu}{\Gamma \cdot \Sigma \vdash c : \nu \cdot \bot}\,(\text{T-Cst})$$

$$\frac{\Gamma \cdot \Sigma \vdash a : \{\ldots, f : \tau, \ldots\}_r \cdot \bot}{\Gamma \cdot \Sigma \vdash a.f : \tau \cdot \bot}\,(\text{T-Fld}) \qquad \frac{\Sigma(\ell) = \rho}{\Gamma \cdot \Sigma \vdash \ell : \rho \cdot \bot}\,(\text{T-Loc})$$

$$\frac{\begin{array}{c}\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \varepsilon_1 \qquad \Gamma[x \mapsto \tau_1] \cdot \Sigma \vdash e_2 : \tau_2 \cdot \varepsilon_2 \\ \forall \rho \in \Gamma(\mathcal{F}_v(e_2) \setminus \{x\}) \cup \Sigma(\mathcal{F}_\ell(e_2)).\ \varepsilon_1 \rhd \rho\end{array}}{\Gamma \cdot \Sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \cdot \varepsilon_1 \sqcup \varepsilon_2}\,(\text{T-Let})$$

$$\frac{\Gamma \cdot \Sigma \vdash a : \texttt{Bool} \cdot \bot \qquad \Gamma \cdot \Sigma \vdash e_1 : \tau \cdot \varepsilon_1 \qquad \Gamma \cdot \Sigma \vdash e_2 : \tau \cdot \varepsilon_2}{\Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \cdot \varepsilon_1 \sqcup \varepsilon_2}\,(\text{T-If})$$

$$\frac{\Gamma \cdot \Sigma \vdash a_i : \tau_i \cdot \bot \qquad \rho = \{f_i : \tau_i{}^{i \in [1,\ldots,n]}\}_r \qquad f_i \text{ are pairwise distinct}}{\Gamma \cdot \Sigma \vdash \{f_i = a_i{}^{i \in [1,\ldots,n]}\} : \rho \cdot (\varnothing \cdot \{\rho\})}\,(\text{T-Alloc})$$

$$\frac{\begin{array}{c}\Gamma \cdot \Sigma \vdash a_i : \rho_i \cdot \bot \qquad \Gamma \cdot \Sigma \vdash a'_{i,j} : \tau'_{i,j} \cdot \bot \qquad \rho_i.f_{i,j} \simeq \tau'_{i,j} \\ \rho_i \text{ are pairwise distinct} \qquad \forall i.\ f_{i,j} \text{ are pairwise distinct} \\ \varphi = \Phi(\rho_i.\{f_{i,j} \leftarrow \tau'_{i,j}{}^{j \in [1,\ldots,k_i]}\}^{i \in [1,\ldots,n]})\end{array}}{\Gamma \cdot \Sigma \vdash a_i.\{f_{i,j} \leftarrow a'_{i,j}{}^{j \in [1,\ldots,k_i]}\}^{i \in [1,\ldots,n]} : \texttt{Unit} \cdot (\{\rho_1, \ldots, \rho_n\} \cdot \varphi)}\,(\text{T-Assign})$$

$$\frac{p : \tau_1 \times \cdots \times \tau_n \to \tau \cdot \varepsilon \qquad \Gamma \cdot \Sigma \vdash a_i : \theta(\tau_i) \cdot \bot}{\Gamma \cdot \Sigma \vdash p(a_1, \ldots, a_n) : \theta(\tau) \cdot \theta(\varepsilon)}\,(\text{T-Call})$$

Fig. 4. Typing rules.

Definition 1. *A type τ is* valid *with respect to effect $(\omega \cdot \varphi)$, written $(\omega \cdot \varphi) \rhd \tau$, if and only if every path from τ to a region in φ passes through a region in ω. Formally, $(\omega \cdot \varphi) \rhd \nu$ is defined inductively by the following rules:*

$$\frac{}{(\omega \cdot \varphi) \rhd \nu} \qquad \frac{\rho \in \omega}{(\omega \cdot \varphi) \rhd \rho} \qquad \frac{\rho \notin \omega \quad \rho \notin \varphi \quad \forall i.\ (\omega \cdot \varphi) \rhd \rho.f_i}{(\omega \cdot \varphi) \rhd \rho}$$

Notice that ω and φ are disjoint, since $(\omega \cdot \varphi)$ is an effect. Consequently, reset regions cannot be valid:

Lemma 1. *For any effect $(\omega \cdot \varphi)$ and any region ρ, $(\omega \cdot \varphi) \rhd \rho \implies \rho \notin \varphi$.*

In the typing rules for let-bindings and conditionals, the overall effect is the union of the effects of sub-expressions, according to the following definition:

Definition 2. *The* union *of two effects $\varepsilon_1 = (\omega_1 \cdot \varphi_1)$ and $\varepsilon_2 = (\omega_2 \cdot \varphi_2)$, denoted $\varepsilon_1 \sqcup \varepsilon_2$, is the pair $(\{\rho \in \omega_1 \mid \varepsilon_2 \rhd \rho\} \cup \{\rho \in \omega_2 \mid \varepsilon_1 \rhd \rho\} \ \cdot\ \varphi_1 \cup \varphi_2)$*

The resulting effect is well-formed, that is, the two sets of regions are disjoint by Lemma 1. Note that the write effect in $\varepsilon_1 \sqcup \varepsilon_2$ is only a subset of $\omega_1 \cup \omega_2$. Indeed, we must take into account that there may be a path from some region ρ in ω_1 to φ_2 that does not pass through ω_2. The existence of such path invalidates ρ.

Therefore, in the definition above the joint write effect is the co-restriction of ω_1 by ε_2 and ω_2 by ε_1.

To provide some intuition behind this, let us consider a conditional expression $\texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. Since we do not know which of the two branches will be realized, the resulting reset effect must be the union of the reset sets of e_1 and e_2. However, we cannot do the same for the write effects. Consider the expression

$$\texttt{if}\ \dots\ \texttt{then}\ \ h_1.\texttt{\{data} \leftarrow h_2.\texttt{data\}}\ \ \texttt{else}\ \ h_2.\texttt{\{data} \leftarrow h_1.\texttt{data\}}$$

where $h_1.\texttt{data}$ and $h_2.\texttt{data}$ are distinct. Let the type of h_1 be $\rho_1 = \texttt{\{data} : \rho_1'\texttt{\}}_{r_1}$ and the type of h_2 be $\rho_2 = \texttt{\{data} : \rho_2'\texttt{\}}_{r_2}$. The effect of the first branch is $(\{\rho_1\} \cdot \{\rho_1', \rho_2'\})$, which invalidates ρ_2. The effect of the second branch is $(\{\rho_2\} \cdot \{\rho_1', \rho_2'\})$, which invalidates ρ_1. Without knowing which branch is going to be executed, we have to invalidate both ρ_1 and ρ_2 after the conditional. We achieve this by removing them from the joint write effect, so that they can no more provide valid access to the reset regions. It may seem that we just lost information about the actual effect of the expression, but for our purposes it does not matter as we prohibit any further mention of h_1, h_2, $h_1.\texttt{data}$, $h_2.\texttt{data}$ anyway.

The lemma below shows that type validity distributes over the effect union.

Lemma 2. *For any ε_1, ε_2, and τ, $\varepsilon_1 \sqcup \varepsilon_2 \rhd \tau$ if and only if $\varepsilon_1 \rhd \rho$ and $\varepsilon_2 \rhd \tau$.*

Effects possess some nice algebraic properties.

Lemma 3. *Effects form a bounded join-semilattice over $\sqcup$ and $\bot$.*

Consequently, we have an order relation on effects as follows:

Definition 3. *We say that an effect ε_1 is a* sub-effect *of ε_2, denoted $\varepsilon_1 \sqsubseteq \varepsilon_2$, when $\varepsilon_2 = \varepsilon_1 \sqcup \varepsilon_2$.*

The T-Alloc rule assigns $\texttt{\{}f_i = a_i{}^{\,i\in[1,\dots,n]}\texttt{\}}$ a region $\rho = \texttt{\{}f_i : \tau_i{}^{\,i\in[1,\dots,n]}\texttt{\}}_r$ where each τ_i matches the type of the corresponding expression a_i. Notice that the index r can be chosen arbitrarily and is not necessarily distinct from the indices of regions in the environments Γ and Σ. Indeed, resetting ρ in the effect for $\texttt{\{}f_i = a_i{}^{\,i\in[1,\dots,n]}\texttt{\}}$ forbids the further use of previous inhabitants of ρ, if any. For instance, in the expression $\texttt{let}\ x = \texttt{\{}f = 41\texttt{\}}\ \texttt{in let}\ y = \texttt{\{}f = 43\texttt{\}}\ \texttt{in}\ e$, the variables x and y can be given two distinct regions and then both can occur in e. It is also possible to give to x and y the same region. In this case, x is invalidated by the second allocation and consequently cannot be used in e. Incidentally, this shows that our type system does not possess the principal type property.

In the T-Assign rule, the operation Φ verifies the validity of an assignment and computes the corresponding reset effect. To define Φ, we shall need several intermediate definitions. Below, we write A to refer to the parameter of Φ, which is the projection of the assignment expression into types: Given a region ρ_i and a field $f_{i,j}$ involved in the assignment, $A(\rho_i, f_{i,j})$ denotes $\tau'_{i,j}$, the type of the value assigned to field $f_{i,j}$. For all ρ and f not affected by A, $A(\rho, f)$ stands for $\rho.f$. We extend this notation to paths as usual: $A(\rho, \epsilon) \triangleq \rho$, $A(\rho, f\pi) \triangleq A(A(\rho, f), \pi)$ when $A(\rho, f)$ is itself a region, and $A(\rho, f\pi) \triangleq A(\rho, f).\pi$ when $A(\rho, f)$ is a scalar type (then π has to be empty). Since the T-Assign rule requires the assigned types to be structurally equal to the original field types, $A(\rho,\pi)$ is defined if and only if $\rho.\pi$ is defined, and if $\rho.\pi$ or $A(\rho,\pi)$ is a scalar type, then $A(\rho,\pi) = \rho.\pi$. We now define a binary relation σ_A as follows:

$$\sigma_A \triangleq \{\, \langle A(\rho,\pi), \rho.\pi \rangle \mid \rho \text{ is affected by } A \text{ and } \rho.\pi \text{ is defined} \,\}$$

An assignment A is *valid* if and only if σ_A is bijective. Intuitively, if σ_A is not bijective, then the assignment contains an alias conflict that cannot be resolved. For instance, Figure 5 represents an assignment where the field h_2 of a region ρ is replaced with a structurally equal region ρ_4. This assignment breaks a former alias between $\rho.h_1.f$ and $\rho.h_2.g$ and thus is invalid. Similarly, Figure 6 shows an assignment that introduces a previously non-existing alias between $\rho.h.f$ and $\rho.h.g$, which is also forbidden.

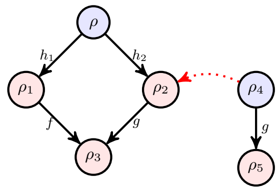

Fig. 5. $\langle \rho_3, \rho_3 \rangle, \langle \rho_5, \rho_3 \rangle \in \sigma_A$.

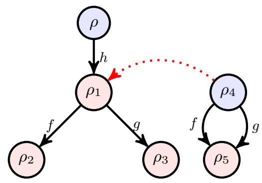

Fig. 6. $\langle \rho_5, \rho_2 \rangle, \langle \rho_5, \rho_3 \rangle \in \sigma_A$.

The reset effect of a valid assignment A, denoted $\Phi(A)$, is defined as follows:

$$\Phi(A) \triangleq \{\, \rho \mid \text{there exists } \rho' \neq \rho \text{ such that } \langle \rho, \rho' \rangle \in \sigma_A \text{ or } \langle \rho', \rho \rangle \in \sigma_A \,\}$$

In other words, we reset every region, on the left or on the right side of an assignment, which is not mapped to itself by σ_A.

In the T-Call rule, we require the type signature of the function to be instantiated with a region substitution. The injectivity property ensures that distinct regions in the signature are instantiated with distinct regions at the call site. Thus, a function verified under certain separation hypotheses is guaranteed to be called in a conforming way.

Typing a computation state. Let us now define what it means for a particular state of computation $\mu \cdot e$ to be well-typed. Since we only evaluate closed program expressions, the variable-typing environment Γ is irrelevant and we omit it below.

Definition 4. *A store μ is* well-typed *in Σ on a set of locations L, denoted $\Sigma \models \mu \cdot L$, if and only if for every location $\ell \in L$ and every path π, either $\Sigma(\ell).\pi = \Sigma(\mu(\ell.\pi))$, or both $\Sigma(\ell).\pi$ and $\mu(\ell.\pi)$ are undefined.*

Definition 5. *A store typing Σ is* injective *on $\mu \cdot L$, denoted $\Sigma \ltimes \mu \cdot L$, if and only if, for any locations $\ell_1, \ell_2 \in L$, and paths π_1, π_2 such that $\Sigma(\ell_1).\pi_1$ and $\Sigma(\ell_2).\pi_2$ are the same region, we have $\mu(\ell_1.\pi_1) = \mu(\ell_2.\pi_2)$.*

We write $\Sigma \models \mu \cdot e$ for $\Sigma \models \mu \cdot \mathcal{F}_\ell(e)$, and $\Sigma \ltimes \mu \cdot e$ for $\Sigma \ltimes \mu \cdot \mathcal{F}_\ell(e)$.

3 Correctness

To demonstrate that our type system is sound and adequate for static control of aliases, we need to show that a single step of execution of a well-typed program preserves the type of the program, the well-typedness of the store, and the injectivity of the store typing.

Theorem 1 (Subject Reduction). *For any reduction step* $\mu \cdot e \longrightarrow \mu' \cdot e'$,

$$\begin{array}{r} \Gamma \cdot \Sigma \vdash e : \tau \cdot \varepsilon \\ \Sigma \models \mu \cdot e \\ \Sigma \ltimes \mu \cdot e \end{array} \quad \Longrightarrow \quad \exists\, \Sigma', \varepsilon', \varphi''. \quad \begin{array}{r} \Gamma \cdot \Sigma' \vdash e' : \tau \cdot \varepsilon' \sqcup (\varnothing \cdot \varphi'') \\ \Sigma' \models \mu' \cdot e' \\ \Sigma' \ltimes \mu' \cdot e' \end{array}$$

where $\varepsilon' \sqsubseteq \varepsilon$ *and* $\varphi'' \cap \Sigma'(\operatorname{dom} \mu') = \varnothing$.

Proof. First of all, let us determine an effect $\varepsilon_0 = (\omega_0 \cdot \varphi_0)$ which is realized during the reduction step together with the "remaining" effect $\varepsilon' = (\omega' \cdot \varphi')$. We can do this by recursion over the derivation of the evaluation step $\mu \cdot e \longrightarrow \mu' \cdot e'$ as follows.

If e is a record allocation or a parallel assignment, then the realized effect ε_0 is simply the effect of e, that is ε, and the remaining effect ε' is empty. Indeed, both expressions reduce to values.

If e is a conditional, then the realized effect ε_0 is empty and the remaining effect is the effect of the chosen branch.

If e is a call to a defined function, then the reduction step consists in definition expansion and does not produce any effect, so that ε_0 is empty. We define the remaining effect ε' to be the effect of the call, ε. Notice that we do not include the additional reset effects of the function body in ε': they will become φ''.

If e is a let-expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, where e_1 is a reducible expression, then ε_0 is the realized effect of e_1 and ε' is $\varepsilon'_1 \sqcup \varepsilon_2$, where ε'_1 is the remaining effect of e_1 and ε_2 is the effect of e_2.

In all other cases, both ε_0 and ε' are empty: indeed, the redex is e itself and does not contain any effects at all.

It is easy to show that $\varepsilon' \sqsubseteq \varepsilon$. Indeed, when we reduce a conditional to one of its branches, $\varepsilon = \varepsilon' \sqcup \hat{\varepsilon}$ where $\hat{\varepsilon}$ is the effect of the discarded branch. When we reduce under a let-expression, $\varepsilon'_1 \sqsubseteq \varepsilon_1$ implies $\varepsilon' = \varepsilon'_1 \sqcup \varepsilon_2 \sqsubseteq \varepsilon_1 \sqcup \varepsilon_2 = \varepsilon$. The other cases are trivial. Similarly, $\varepsilon_0 \sqsubseteq \varepsilon$.

We can also see that for any location $\ell \in \operatorname{dom} \mu$ that appears in e', $\varepsilon_0 \triangleright \Sigma(\ell)$. Indeed, if e is a record allocation, then the resulting location is not in μ. If e is an assignment, then it reduces to the unit constant `()` which does not contain any locations. Finally, if e is $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ with reducible e_1, then every location in e_2 has a valid type with respect to ε_1. Since $\varepsilon_0 \sqsubseteq \varepsilon_1$, we obtain $\varepsilon_0 \triangleright \Sigma(\ell)$ by Lemma 2. In all other cases, ε_0 is empty and the claim is trivial.

Let us now define the new store typing Σ'. If the redex sub-expression is a parallel assignment, we consider its typing derivation and the corresponding relation σ_A. Then for every location ℓ,

$$\Sigma'(\ell) \triangleq \begin{cases} \rho & \text{if } \langle \Sigma(\ell), \rho \rangle \in \sigma_A \\ \Sigma(\ell) & \text{otherwise.} \end{cases}$$

If the redex is a record allocation of type ρ reduced to a fresh location ℓ, then $\Sigma' \triangleq \Sigma[\ell \mapsto \rho]$. If the redex is neither an assignment nor an allocation, $\Sigma' \triangleq \Sigma$.

Notice that for any location $\ell \in \operatorname{dom} \mu$, if $\Sigma'(\ell) \neq \Sigma(\ell)$ then both $\Sigma(\ell)$ and $\Sigma'(\ell)$ are in φ_0 by definition of $\Phi(A)$. Consequently, for every $\ell \in \operatorname{dom} \mu$ that appears in e', we have $\Sigma'(\ell) = \Sigma(\ell)$, due to $\varepsilon_0 \triangleright \Sigma(\ell)$ and Lemma 1.

Before we proceed, we need to establish a variant of the frame property, namely that all "observable" store modifications and region resets are covered by the realized write effect ω_0.

Lemma 4. *Let ℓ be a location in e'. Let π be a path such that $\mu(\ell.\pi)$ is defined and for every proper prefix $\bar{\pi} \prec \pi$, $\Sigma(\ell).\bar{\pi}$ is not in ω_0. Then $\Sigma(\ell).\pi \notin \varphi_0$ and $\mu'(\ell.\pi) = \mu(\ell.\pi)$.*

Proof. We proceed by induction on π. For $\pi = \epsilon$, we obtain $\Sigma(\ell).\epsilon = \Sigma(\ell) \notin \varphi_0$. We also have $\mu'(\ell.\epsilon) = \mu(\ell.\epsilon) = \ell$, since reduction rules do not remove locations from the store. Now, let π be a non-empty path $\pi' f$ such that $\mu(\ell.\pi)$ is defined and for all $\bar{\pi} \prec \pi$, $\Sigma(\ell).\bar{\pi} \notin \omega_0$. Since $\varepsilon_0 \triangleright \Sigma(\ell)$ and no region on the path from $\Sigma(\ell)$ to $\Sigma(\ell).\pi$ is in ω_0, we obtain $\Sigma(\ell).\pi \notin \varphi_0$. By induction hypothesis, $\mu'(\ell.\pi') = \mu(\ell.\pi')$. Since nothing was written into the field f of $\mu(\ell.\pi')$ during the reduction step (otherwise, $\Sigma(\mu(\ell.\pi')) = \Sigma(\ell).\pi'$ would appear either in ω_0 or in φ_0), we conclude that $\mu'(\ell.\pi) = \mu(\ell.\pi)$. ☐

Now we are ready to attack the main theorem. We prove the desired properties by induction over the derivation of the reduction step. Looking at the last reduction rule in the derivation, we have four interesting cases.

Case E-CTX. Assume e is of the form `let` $x = e_1$ `in` e_2 and e_1 reduces to e_1'. For every location ℓ in e_2, $\Sigma'(\ell) = \Sigma(\ell)$, since $\ell \in \operatorname{dom} \mu$ and occurs in e'.

Type preservation. By induction hypothesis on $\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'$, we have $\Gamma \cdot \Sigma' \vdash e_1' : \tau_1 \cdot \varepsilon_1' \sqcup (\varnothing \cdot \varphi'')$, where τ_1 is the type of e_1, ε_1' is the remaining effect of e_1, and $\varphi'' \cap \Sigma'(\operatorname{dom} \mu') = \varnothing$. Notice that Σ' is the same as defined above, since it only depends on the redex expression inside e_1'.

Let τ_2 and ε_2 be, respectively, the type and the effect of e_2 in the typing derivation for e. Since Σ' coincides with Σ on every location in e_2, we obtain $\Gamma[x \mapsto \tau_1] \cdot \Sigma' \vdash e_2 : \tau_2 \cdot \varepsilon_2$.

Let us now show that for every location ℓ in e_2, we have $\varepsilon_1' \sqcup (\varnothing \cdot \varphi'') \triangleright \Sigma'(\ell)$. First, we know that $\varepsilon_1' \triangleright \Sigma'(\ell)$. Indeed, since e is well-typed and $\Sigma'(\ell) = \Sigma(\ell)$, we have $\varepsilon_1 \triangleright \Sigma'(\ell)$. Since $\varepsilon_1' \sqsubseteq \varepsilon_1$, we obtain $\varepsilon_1' \triangleright \Sigma'(\ell)$ by Lemma 2. Furthermore, $\varphi'' \cap \Sigma'(\operatorname{dom} \mu') = \varnothing$ implies $(\varnothing \cdot \varphi'') \triangleright \Sigma'(\ell)$. Since e is closed, $\mathcal{F}_v(e_2) \setminus \{x\}$ is empty, and Lemma 2 gives us the third premise of the T-LET rule for e'. Finally, by Lemma 3, $\varepsilon_1' \sqcup (\varnothing \cdot \varphi'') \sqcup \varepsilon_2 = \varepsilon' \sqcup (\varnothing \cdot \varphi'')$. Altogether, we obtain $\Gamma \cdot \Sigma' \vdash e' : \tau \cdot \varepsilon' \sqcup (\varnothing \cdot \varphi'')$.

Well-typedness of μ'. By induction hypothesis, $\Sigma' \models \mu' \cdot e_1'$. Let ℓ be a location in e_2 and π, an arbitrary path. We need to show that either $\Sigma'(\ell).\pi = \Sigma'(\mu'(\ell.\pi))$ or both are undefined.

Let π' be the longest prefix of π such that $\mu(\ell.\pi')$ is defined and for all $\bar{\pi} \prec \pi'$, $\Sigma(\ell).\bar{\pi} \notin \omega_0$. Such a prefix exists, since $\Sigma \models \mu \cdot e$ and therefore $\mu(\ell.\epsilon)$ is defined. By Lemma 4, $\Sigma(\ell).\pi' \notin \varphi_0$ and $\mu'(\ell.\pi') = \mu(\ell.\pi')$. Since $\Sigma(\ell).\pi' = \Sigma(\mu(\ell.\pi'))$ (by well-typedness of μ), $\Sigma(\ell).\pi' \notin \varphi_0$ implies $\Sigma(\mu(\ell.\pi')) = \Sigma'(\mu(\ell.\pi'))$. Overall, we obtain $\Sigma'(\ell).\pi' = \Sigma(\ell).\pi' = \Sigma(\mu(\ell.\pi')) = \Sigma'(\mu(\ell.\pi')) = \Sigma'(\mu'(\ell.\pi'))$.

Now we have three cases to consider. If $\pi' = \pi$, then $\Sigma'(\ell).\pi = \Sigma'(\mu'(\ell.\pi))$. If $\pi' \prec \pi$ and for every path $\bar{\pi}$ such that $\pi' \prec \bar{\pi} \preceq \pi$, $\mu(\ell.\bar{\pi})$ is undefined, then $\Sigma'(\ell).\pi = \Sigma(\ell).\pi$ is undefined by well-typedness of μ, and $\mu'(\ell.\pi)$ is undefined, since $\mu'(\ell.\pi') = \mu(\ell.\pi')$.

Otherwise, $\Sigma(\ell).\pi' \in \omega_0$. Then the redex expression is an assignment, and we can consider the corresponding relation σ_A. Let ℓ' denote the location $\mu(\ell.\pi')$. Since μ is well-typed and Σ is injective on $\mu \cdot e$, no location in e other than ℓ' can have type $\Sigma(\ell') = \Sigma(\ell).\pi'$. Consequently, we know that the store is modified at ℓ' at this reduction step.

Let $\pi = \pi'\pi''$. If $\Sigma(\ell').\pi''$ is defined, then $\langle A(\Sigma(\ell'), \pi''), \Sigma(\ell').\pi'' \rangle$ is in σ_A. The first component is $\Sigma(\mu'(\ell'.\pi''))$, the Σ-type of the value found in the store at $\ell'.\pi''$ after the assignment. The second component is $\Sigma'(\mu'(\ell'.\pi''))$, the type given to this value in Σ'. Notice that $\Sigma(\ell').\pi'' = A(\Sigma(\ell'), \pi'') = \Sigma(\mu'(\ell'.\pi'')) = \Sigma'(\mu'(\ell'.\pi''))$ when $\mu'(\ell'.\pi'')$ is a scalar. We get $\Sigma'(\ell).\pi = \Sigma(\ell).\pi = \Sigma(\ell').\pi'' = \Sigma'(\mu'(\ell'.\pi'')) = \Sigma'(\mu'(\mu(\ell.\pi').\pi'')) = \Sigma'(\mu'(\mu'(\ell.\pi').\pi'')) = \Sigma'(\mu'(\ell.\pi))$.

If $\Sigma'(\ell).\pi = \Sigma(\ell).\pi = \Sigma(\ell').\pi''$ is undefined, then $\mu(\ell'.\pi'')$ is also undefined. In a parallel assignment, the types of the affected fields are structurally equal to the types of the corresponding assigned values. Since μ is well-typed, this means that ℓ' in μ (before the reduction step) admits exactly the same paths as in μ' (after the reduction step). Therefore, $\mu'(\ell'.\pi'') = \mu'(\ell.\pi)$ is undefined.

Injectivity of Σ'. Consider ℓ_1 and ℓ_2 in e' and two paths π_1 and π_2 such that $\Sigma'(\ell_1).\pi_1$ and $\Sigma'(\ell_2).\pi_2$ are the same region. We need to prove that $\mu'(\ell_1.\pi_1) = \mu'(\ell_2.\pi_2)$. We have three cases to consider.

If neither ℓ_1 nor ℓ_2 are in μ, then they are allocated during the reduction step, and therefore both appear in e_1'. Then we conclude by induction hypothesis.

Assume $\ell_1 \notin \operatorname{dom} \mu$ and $\ell_2 \in \operatorname{dom} \mu$ (the symmetric case is handled in the same way). Then the redex is a record allocation and ℓ_1 is the new location. This implies that $\omega_0 = \varnothing$ and $\Sigma'(\ell_1) \in \varphi_0$. Since μ is well-typed, $\mu(\ell_2.\pi_2)$ is defined, and we get $\Sigma'(\ell_2).\pi_2 = \Sigma(\ell_2).\pi_2 \notin \varphi_0$ and $\mu'(\ell_2.\pi_2) = \mu(\ell_2.\pi_2)$ by Lemma 4. Therefore π_1 can not be ϵ. Let $\pi_1 = f\pi'$. Then there exists a location ℓ' in e_1 such that $\Sigma'(\ell_1).f = \Sigma(\ell')$ and $\mu'(\ell_1.f) = \ell'$. By injectivity of Σ, $\mu(\ell'.\pi') = \mu(\ell_2.\pi_2)$. Moreover, since record allocation does not modify the existing entries in the store, $\mu'(\ell'.\pi') = \mu(\ell'.\pi')$, and we obtain $\mu'(\ell_1.\pi_1) = \mu'(\ell_2.\pi_2)$.

Now, let both ℓ_1 and ℓ_2 be in μ. Since $\Sigma'(\ell_1) = \Sigma(\ell_1)$ and $\Sigma'(\ell_2) = \Sigma(\ell_2)$, we have $\mu(\ell_1.\pi_1) = \mu(\ell_2.\pi_2)$ by injectivity of Σ. Indeed, if ℓ_1 is in e, the injectivity applies immediately. If ℓ_1 is not in e, then it could only appear in e' after reducing some $\ell_0.f$ in e, in which case we apply injectivity to $\ell_0.f\pi_1$ and $\ell_2.\pi_2$.

Let π_1' be the longest prefix of π_1 such that for all $\bar\pi \prec \pi_1'$, $\Sigma(\ell).\bar\pi \notin \omega_0$. Let π_2' be the longest prefix of π_2 such that for all $\bar\pi \prec \pi_2'$, $\Sigma(\ell).\bar\pi \notin \omega_0$. By Lemma 4, we have $\Sigma(\ell_1).\pi_1' \notin \varphi_0$, $\Sigma(\ell_2).\pi_2' \notin \varphi_0$, $\mu'(\ell_1.\pi_1') = \mu(\ell_1.\pi_1')$, and $\mu'(\ell_2.\pi_2') = \mu(\ell_2.\pi_2')$.

If $\pi_1' = \pi_1$ and $\pi_2' = \pi_2$, then $\mu'(\ell_1.\pi_1) = \mu'(\ell_2.\pi_2)$ immediately. Otherwise, $\Sigma(\ell_1).\pi_1'$ or $\Sigma(\ell_2).\pi_2'$ or both are in ω_0. This means that the redex expression is an assignment, and we can consider the corresponding relation σ_A.

Assume $\Sigma(\ell_1).\pi_1' \in \omega_0$. Let ℓ_1' be $\mu(\ell_1.\pi_1')$ and $\pi_1 = \pi_1'\pi_1''$. Then σ_A contains a pair $\langle A(\Sigma(\ell_1'), \pi_1''), \Sigma(\ell_1').\pi_1''\rangle$. The first component is $\Sigma(\mu'(\ell_1'.\pi_1''))$, the Σ-type of the location found in the store at $\ell_1'.\pi_1''$ after the assignment. The second component is $\Sigma'(\mu'(\ell_1'.\pi_1''))$, the type given to this location in Σ'. Once again, we have three cases to consider.

If $\pi_2' = \pi_2$ and $A(\Sigma(\ell_1'), \pi_1'') \neq \Sigma(\ell_1').\pi_1''$, then $\Sigma(\ell_1').\pi_1'' \in \varphi_0$ by definition of $\Phi(A)$. Since $\varepsilon_0 \triangleright \Sigma(\ell_2)$ and $\Sigma(\ell_2).\pi_2 = \Sigma(\ell_1).\pi_1 = \Sigma(\ell_1').\pi_1''$ is in φ_0, there must be some $\bar\pi \prec \pi_2$ such that $\Sigma(\ell_2).\bar\pi \in \omega_0$, which contradicts the definition of π_2'.

If $\pi_2' = \pi_2$ and $\Sigma(\mu'(\ell_1'.\pi_1'')) = A(\Sigma(\ell_1'), \pi_1'') = \Sigma(\ell_1').\pi_1'' = \Sigma(\mu(\ell_1', \pi_1''))$, then $\mu'(\ell_1'.\pi_1'') = \mu(\ell_1', \pi_1'')$ by injectivity of Σ. Indeed, the location $\mu'(\ell_1'.\pi_1'')$ is reachable in μ from the right-hand side of the reduced assignment in e. We obtain $\mu'(\ell_1.\pi_1) = \mu'(\mu'(\ell_1.\pi_1').\pi_1'') = \mu'(\mu(\ell_1.\pi_1').\pi_1'') = \mu'(\ell_1'.\pi_1'') = \mu(\ell_1', \pi_1'') = \mu(\ell_1, \pi_1) = \mu(\ell_2, \pi_2) = \mu'(\ell_2, \pi_2)$.

Otherwise, $\pi_2' \prec \pi_2$ and $\Sigma(\ell_2).\pi_2' \in \omega_0$. Let ℓ_2' be $\mu(\ell_2.\pi_2')$ and $\pi_2 = \pi_2'\pi_2''$. Then σ_A contains $\langle A(\Sigma(\ell_2'), \pi_2''), \Sigma(\ell_2').\pi_2''\rangle$. Since $\Sigma(\ell_1').\pi_1'' = \Sigma(\ell_2').\pi_2''$ and σ_A is a bijection, we have $\Sigma(\mu'(\ell_1'.\pi_1'')) = A(\Sigma(\ell_1'), \pi_1'') = A(\Sigma(\ell_2'), \pi_2'') = \Sigma(\mu'(\ell_2'.\pi_2''))$. Both $\mu'(\ell_1'.\pi_1'')$ and $\mu'(\ell_2'.\pi_2'')$ are reachable in μ from the reduced assignment in e, and we can conclude by injectivity of Σ that they are actually the same location. We obtain $\mu'(\ell_1.\pi_1) = \mu'(\ell_1'.\pi_1'') = \mu'(\ell_2'.\pi_2'') = \mu'(\ell_2, \pi_2)$.

Case E-ALLOC. Let e be a record allocation. Then the type of e is some region ρ and e' is a fresh location ℓ. Type preservation is trivial, since $\Sigma'(\ell) = \rho$ by construction, $\varepsilon' = \bot$, and we set $\varphi'' = \varnothing$.

We now show that μ' is well-typed. Let π be an arbitrary path. If $\pi = \epsilon$ then $\mu'(\ell.\epsilon) = \ell$ and thus $\Sigma'(\ell).\epsilon = \Sigma'(\mu'(\ell.\epsilon))$. Let $\pi = f\pi'$. If there is no field f in the record e, then $\mu'(\ell.\pi)$ and $\Sigma'(\ell).\pi = \rho.\pi$ are both undefined. If f is initialized in e with a scalar constant c of type ν, then $\mu'(\ell.f\pi')$ and $\Sigma'(\ell).f\pi'$ are only defined when $\pi' = \epsilon$, in which case $\Sigma'(\ell).f = \nu = \Sigma'(c) = \Sigma'(\mu'(\ell.f))$. Finally, if f is initialized in e with a location ℓ', then $\Sigma(\ell').\pi' = \Sigma(\mu(\ell'.\pi'))$ (or both are undefined) by well-typedness of μ. Then we have $\Sigma'(\ell).\pi = \rho.f\pi' = \Sigma(\ell').\pi' = \Sigma(\mu(\ell'.\pi')) = \Sigma'(\mu(\ell'.\pi')) = \Sigma'(\mu'(\ell'.\pi')) = \Sigma'(\mu'(\ell.\pi))$.

As for injectivity of Σ', since ℓ is the only location in e', it is enough to take two distinct paths π_1 and π_2 such that $\rho.\pi_1 = \rho.\pi_2$. Neither of two paths can be empty, as ρ cannot be equal to a part of ρ. Assuming $\pi_1 = f_1\pi_1'$ and $\pi_2 = f_2\pi_2'$, we obtain $(\rho.f_1).\pi_1' = (\rho.f_2).\pi_2'$, where $\rho.f_1 = \Sigma(\ell_1')$ and $\rho.f_2 = \Sigma(\ell_2')$ for some ℓ_1' and ℓ_2' occurring in e. Then we can use the injectivity of Σ and obtain $\mu'(\ell.\pi_1) = \mu(\ell_1'.\pi_1') = \mu(\ell_2'.\pi_2') = \mu'(\ell.\pi_2)$.

Case E-ASSIGN. Let e be an assignment. Then the type of e is `Unit`, e' is `()`, $\varepsilon' = \bot$, and $\varphi'' = \varnothing$. The claim trivially holds, in particular, since $\mathcal{F}_\ell(e') = \varnothing$.

Case E-δ. Let e be of the form $p(v_1, \dots, v_n)$ with $p(x_1, \dots, x_n) \mapsto \hat{e}$. Then the reduct e' is $\hat{e}[x_i/v_i \ ^{i\in\{1,\dots,n\}}]$. Since $\mathcal{F}_\ell(\hat{e}) = \varnothing$, all locations in e' occur in e. Moreover, the reduction step does not modify the store and $\Sigma' = \Sigma$. Consequently, the well-typedness of μ' and injectivity of Σ' trivially hold.

Let us prove the type preservation. Let $\hat{\tau}_1 \times \ldots \times \hat{\tau}_n \to \hat{\tau} \cdot (\hat{\omega} \cdot \hat{\varphi})$ be the signature of p. By hypothesis, $\Gamma[x_i \mapsto \hat{\tau}_i \ ^{i\in\{1,\dots,n\}}] \cdot \Sigma \vdash \hat{e} : \hat{\tau} \cdot (\hat{\omega} \cdot \hat{\varphi} \cup \hat{\varphi}'')$ where $\hat{\varphi}''$ is disjoint with $\mathcal{R}(\hat{\tau}_1, \dots, \hat{\tau}_n, \hat{\tau})$. It is easy to show that every region that occurs in the type inference for $\hat{e}$ belongs one of these two sets. Indeed, every such region either comes from the type of a formal parameter, or is introduced through a record allocation or a function call, and therefore is reset.

Since e is well-typed, there exists a region substitution θ that maps $\hat{\tau}$ to τ, $(\hat{\omega} \cdot \hat{\varphi})$ to ε, and each $\hat{\tau}_i$ to $\Sigma(v_i)$. Let θ' be an extension of θ where every region of φ'' is mapped to a fresh region in such a way that θ' remains a region substitution, that is injective and consistent with respect to sub-regions. Now, we can apply θ' throughout the type inference tree for $\hat{e}$, and obtain a valid type judgement $\Gamma[x_i \mapsto \Sigma(v_i) \ ^{i\in\{1,\dots,n\}}] \cdot \Sigma \vdash \hat{e} : \tau \cdot (\omega \cdot \varphi \cup \theta'(\hat{\varphi}''))$ where $(\omega \cdot \varphi) = \epsilon$. We define φ'' to be $\theta'(\hat{\varphi}'')$. It is easy to see that $(\omega \cdot \varphi \cup \varphi'') = (\omega \cdot \varphi) \sqcup (\varnothing \cdot \varphi'')$ and $\varphi'' \cap \Sigma'(\operatorname{dom} \mu') = \varnothing$, since every region in φ'' is fresh.

Finally, by standard substitution lemma ([8, p.106]), replacing formal parameters $x_1, \dots, x_n$ with the corresponding argument values v_i in $\hat{e}$ results in a valid typing judgement $\Gamma \cdot \Sigma' \vdash \hat{e}[x_i/v_i \ ^{i\in\{1,\dots,n\}}] : \tau \cdot \varepsilon \sqcup (\varnothing \cdot \varphi'')$.

In all other cases, the reduction step does not produce any effect ($\varepsilon_0 = \bot$), the store and the store typing do not change, and all locations in e' are accessible from locations in e. Type preservation can then be proved in a usual way, and the two other properties are trivial. □

4 Related Work

Our approach is to track and control aliasing statically, using a type system with regions and effects. This methodology originates from the work of Baker [9], Lucassen and Gifford [10], and region-based memory management of Tofte and Talpin [11]. In their work, regions are used to ensure that, while two pointers inside the same region may or may not be aliased, two pointers belonging to distinct regions are never aliased. That is, regions can be thought of as equivalence classes of pointers for a statically known, approximative "may alias" relation. In our case, though, a region does not denote a set of memory locations, but a single location. This allows us to describe statically the exact shape of the memory store, two symbolic names being aliased if and only if they are assigned the same region. This is similar to how pointer identity is encoded in *alias types* [12] and *typed regions* [13]. However, both approaches rely on strong updates, which, in the case of alias types, imposes limitations on the control flow or, in the case of typed regions, introduces the complex machinery of dependent types.

Shape analysis [14,15] provides techniques similar to ours for automatically inferring store invariants. For instance, "must-alias" analysis described in [16,17] is based on access-path tracking, where a store location is characterized using a set of paths leading to it from the program variables. For the purposes of verification, however, we cannot afford over-approximations of alias relations and we use the reset effect to maintain the exact representation of the store. The prize to pay is that we have to reject some data type definitions and programs.

In the context of object-oriented programming, *ownership* [18,19,20] techniques and similar type-based approaches such as *islands* [21], *balloon types* [22], and *universe types* [23] provide a methodology for controlled aliasing and alias protection. For instance, the owners-as-dominators paradigm requires that all external accesses to internals of an object must go via its owner's interface. The validity constraint we generate in the typing rule for $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ can be seen as the fact that in e_2 the "owners" of the reset regions of e_1 are exactly regions that can access them only by passing through a region of the write effect of e_1. However, this is merely an analogy and our approach is in fact orthogonal to ownership. The principal goal of ownership types is not to achieve a precise heap description. It rather serves to guarantee a strong notion of encapsulation based on user-provided type annotations that determine which parts of an object are accessible to other objects and when an object can be passed to other objects.

Our *reset* effect has some connections with the concept of *unique variable* [24,21,25]. A unique variable is either null or refers to some unshared object. In our case, the effect for a record allocation prohibits all existing names that refer to the region of the new record. Assuming this record is bound to a variable x, our system makes x a unique variable. This is very similar to Boyland's "alias burying" [26]. In Boyland's work, when a field annotated as unique is read, all existing aliases are required to be dead and will never be used again.

Using effects to describe not just store modifications but also accessibility constraints gives to our type system some flavor of *capabilities* [27,28]. Rather than passing linear capability tokens between producers and consumers, our reset effect can be seen as permission revoking. Moreover, as in the case of alias types, systems with capabilities and permissions rely on strong updates.

5 Conclusion and Perspectives

The proposed approach to alias control is implemented in Why3 [29], a platform for deductive verification. In addition to what is presented in this paper, the implementation also features type- and region-polymorphism, type and region inference, ghost code [30], algebraic data types, and abstract data types. Thanks to region inference, users never have to manipulate regions explicitly.

We intend to extend this type system with the ability to refine a data type by adding new fields and glue invariants. In particular, this will allow users to refine interfaces into implementations, to prove the latter correct.

References

1. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (October 1969) 576–580 and 583

2. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence **7** (1972) 23–50
3. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17th Annual IEEE Symposium on Logic in Computer Science, IEEE Comp. Soc. Press (2002)
4. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Misra, J., Nipkow, T., Sekerinski, E., eds.: 14th International Symposium on Formal Methods (FM'06). Volume 4085 of Lecture Notes in Computer Science., Hamilton, Canada (2006) 268–283
5. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In Felleisen, M., Gardner, P., eds.: Proceedings of the 22nd European Symposium on Programming. Volume 7792 of Lecture Notes in Computer Science., Springer (March 2013) 125–128
6. Berdine, J., O'Hearn, P.W.: Strong update, disposal, and encapsulation in bunched typing. Electr. Notes Theor. Comput. Sci. **158** (2006) 81–98
7. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. SIGPLAN Not. **28**(6) (June 1993) 237–247
8. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
9. Baker, H.G.: Unify and conquer. In: LISP and Functional Programming. (1990) 218–226
10. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '88, New York, NY, USA, ACM (1988) 47–57
11. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation (1997)
12. Smith, F., Walker, D., Morrisett, J.G.: Alias types. In Smolka, G., ed.: Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings. Volume 1782 of Lecture Notes in Computer Science., Springer (2000) 366–381
13. Monnier, S.: Typed regions. In: Second workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE'2004), Venice, Italy (January 2004)
14. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems **24**(3) (2002) 217–298
15. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. SIGPLAN Not. **40**(1) (January 2005) 310–323
16. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol. **17**(2) (May 2008) 9:1–9:34
17. Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., Yahav, E.: Alias analysis for object-oriented programs. [31] 196–232
18. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. [31] 15–58
19. Mycroft, A., Voigt, J.: Notions of aliasing and ownership. [31] 59–83
20. Dietl, W., Müller, P.: Object ownership in program verification. [31] 289–318
21. Hogg, J.: Islands: Aliasing protection in object-oriented languages. SIGPLAN Not. **26**(11) (November 1991) 271–285
22. Almeida, P.S.: Balloon types: Controlling sharing of state in data types. In: Proceedings ECOOP'97. Volume 1241 of LNCS., Springer (June 1997) 32–59

23. Müller, P., Rudich, A.: Ownership transfer in universe types. In: ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA), ACM (2007) 461–478
24. Wadler, P.: Linear types can change the world! In: Programming Concepts and Methods, North (1990)
25. Baker, H.G.: "Use-once" variables and linear objects: Storage management, reflection and multi-threading. SIGPLAN Not. **30**(1) (January 1995) 45–52
26. Boyland, J.: Alias burying: unique variables without destructive reads. j-SPE **31**(6) (May 2001) 533–553
27. Crary, K., Walker, D., Morrisett, G.: Typed memory management in a calculus of capabilities. In: ACM Symposium on Principles of Programming Languages (POPL), ACM Press (1999) 262–275
28. Charguéraud, A., Pottier, F.: Functional translation of a calculus of capabilities. In: ACM SIGPLAN International Conference on Functional Programming (ICFP). (September 2008) 213–224
29. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011) 53–64
30. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. In Biere, A., Bloem, R., eds.: 26th International Conference on Computer Aided Verification. Volume 8859 of Lecture Notes in Computer Science., Vienna, Austria, Springer (July 2014) 1–16
31. Clarke, D., Noble, J., Wrigstad, T., eds.: Aliasing in Object-Oriented Programming: Types, Analysis and Verification. Volume 7850 of Lecture Notes in Computer Science., Springer (2013)

A Proofs of Lemmas

Lemma 1. *For any effect* $(\omega \cdot \varphi)$ *and any region* ρ, $(\omega \cdot \varphi) \triangleright \rho \implies \rho \notin \varphi$.

Proof. Let ρ be a region, and $(\omega \cdot \varphi)$ be an effect such that $(\omega \cdot \varphi) \triangleright \rho$. Either $\rho \in \omega$, then the invariant on effects (disjointness of write and reset effects) ensures that $\rho \notin \varphi$. Otherwise, $\rho \notin \omega$, so $\rho \notin \varphi$ by the inference rule premise. □

Lemma 2. *For any* ε_1, ε_2, *and* τ, $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \tau$ *if and only if* $\varepsilon_1 \triangleright \rho$ *and* $\varepsilon_2 \triangleright \tau$.

Proof. Let τ be a type, and $\varepsilon_1 = (\omega_1 \cdot \varphi_1)$, $\varepsilon_2 = (\omega_2 \cdot \varphi_2)$ a two effects. Below we note $(\omega \cdot \varphi)$ for the union $\varepsilon_1 \sqcup \varepsilon_2$.

Let us first prove that $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \tau$ implies $\varepsilon_1 \triangleright \rho$ and $\varepsilon_2 \triangleright \tau$. We proceed by induction on the derivation of $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \tau$. If $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \nu$, the result trivially holds.

Assume now $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \rho$ with $\rho \in \omega$. We have two sub-cases to consider. Either $\rho \in \omega_1$ and $\varepsilon_2 \triangleright \rho$, in which case we derive $\varepsilon_1 \triangleright \rho$ by a second clause in the definition 1, so the result holds. Otherwise, $\rho \in \omega_2$ and $\varepsilon_1 \triangleright \rho$, in which case we derive $\varepsilon_2 \triangleright \rho$ alike and again the result holds.

Finally, assume $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \rho$ with $\rho \notin \omega$, $\rho \notin \varphi$, and $\forall i. \varepsilon_1 \sqcup \varepsilon_2 \triangleright \rho.f_i$. Since ω_1 and φ_1 are subsets of ω, φ respectively, $\rho \notin \omega_1$ and $\rho \notin \varphi_1$. Moreover, by induction hypothesis, we have $\varepsilon_1 \triangleright \rho.f_i$, so $\varepsilon_1 \triangleright \rho$ holds by the third inference rule in the definition 1. By the similar reasoning, $\varepsilon_2 \triangleright \rho$ also holds, which allows us to conclude.

Let us now prove the other direction. Assume that $(\omega_1 \cdot \varphi_1) \triangleright \tau$ and $(\omega_2 \cdot \varphi_2) \triangleright \tau$. We proceed by induction on $(\omega_1 \cdot \varphi_1) \triangleright \tau$. If $\tau = \nu$, the result trivaily holds.

Assume now that $(\omega_1 \cdot \varphi_1) \triangleright \rho$ and $\rho \in \omega_1$. Then $\rho \in \{\hat{\rho} \in \omega_1 \,|\, \varepsilon_2 \triangleright \hat{\rho}\} \subseteq \omega$, so we get $(\omega \cdot \varphi) \triangleright \rho$ using the second inference rule.

Otherwise, $(\omega_1 \cdot \varphi_1) \triangleright \rho$ and $\rho \notin \omega_1$, $\rho \notin \varphi_1$, and $\forall i. (\omega_1 \cdot \varphi_1) \triangleright \rho.f_i$. We have two sub-cases to consider. Either $(\omega_2 \cdot \varphi_2) \triangleright \rho$ holds with $\rho \in \omega_2$. Then $\rho \in \{\hat{\rho} \in \omega_2 \,|\, \varepsilon_1 \triangleright \hat{\rho}\} \subseteq \omega$, so again we get $(\omega \cdot \varphi) \triangleright \rho$ using the second inference rule. Otherwise, $(\omega_2 \cdot \varphi_2) \triangleright \rho$ holds with $\rho \notin \omega_2$, $\rho \notin \varphi_2$, and $\forall i. (\omega_2 \cdot \varphi_2) \triangleright \rho.f_i$. We can thus apply the induction hypothesis on each $(\omega_1 \cdot \varphi_1) \triangleright \rho.f_i$ and get $\forall i. (\omega \cdot \varphi) \triangleright \rho.f_i$. Moreover, $\rho \notin \omega$ and $\rho \notin \varphi$, so we get $(\omega \cdot \varphi) \triangleright \rho$ using the third inference rule. $\square$

Lemma 3. *Effects form a bounded join-semilattice over $\sqcup$ and $\bot$.*

Proof. Clearly, for any effect ε, $\varepsilon \sqcup \bot = \bot \sqcup \varepsilon = \varepsilon$, i.e., $\bot$ is the identity element. Moreover,

- $\sqcup$ is idempotent. Indeed $\omega = \{\rho \in \omega \,|\, (\omega \cdot \varphi) \triangleright \rho\}$, so $(\omega \cdot \varphi) \sqcup (\omega \cdot \varphi) = (\omega \cdot \varphi)$.
- $\sqcup$ is commutative. Indeed, the union of $(\omega_1 \cdot \varphi_1)$ and $(\omega_2 \cdot \varphi_2)$, is defined by $(\{\rho \in \omega_1 \,|\, \varepsilon_2 \triangleright \rho\} \cup \{\rho \in \omega_2 \,|\, \varepsilon_1 \triangleright \rho\} \ \cdot \ \varphi_1 \cup \varphi_2)$ which is obviously equal to $(\{\rho \in \omega_2 \,|\, \varepsilon_1 \triangleright \rho\} \cup \{\rho \in \omega_1 \,|\, \varepsilon_2 \triangleright \rho\} \ \cdot \ \varphi_2 \cup \varphi_1)$.
- $\sqcup$ is associative. Let us denote $\{\rho \in \omega \,|\, \varepsilon \triangleright \rho\}$ shortly by $\omega \,|\, \varepsilon$. First, observe that for any ω, ε, and ε', by Lemma 2, $(\omega \,|\, \varepsilon) \,|\, \varepsilon' = \omega \,|\, \varepsilon \sqcup \varepsilon' = (\omega \,|\, \varepsilon') \,|\, \varepsilon$. Therefore, for any ε_1, ε_2, ε_3 we have

$$
\begin{array}{lr}
\quad (\varepsilon_1 \sqcup \varepsilon_2) \sqcup \varepsilon_3 & \\
= (\omega_1 \,|\, \varepsilon_2 \,\cup\, \omega_2 \,|\, \varepsilon_1 \cdot \varphi_1 \cup \varphi_2) \sqcup \varepsilon_3 & (\text{def}) \\
= ((\omega_1 \,|\, \varepsilon_2 \,\cup\, \omega_2 \,|\, \varepsilon_1) \,|\, \varepsilon_3 \cup \omega_3 \,|\, \varepsilon_1 \sqcup \varepsilon_2 \cdot \varphi_1 \cup \varphi_2 \cup \varphi_3) & (\text{def}) \\
= ((\omega_1 \,|\, \varepsilon_2) \,|\, \varepsilon_3 \cup\, (\omega_2 \,|\, \varepsilon_1) \,|\, \varepsilon_3 \cup \omega_3 \,|\, \varepsilon_1 \sqcup \varepsilon_2 \cdot \varphi_1 \cup \varphi_2 \cup \varphi_3) & \\
= (\omega_1 \,|\, \varepsilon_2 \sqcup \varepsilon_3 \,\cup\, (\omega_2 \,|\, \varepsilon_3) \,|\, \varepsilon_1 \,\cup\, (\omega_3 \,|\, \varepsilon_2) \,|\, \varepsilon_1 \cdot \varphi_1 \cup \varphi_2 \cup \varphi_3) & \\
= (\omega_1 \,|\, \varepsilon_2 \sqcup \varepsilon_3 \,\cup\, (\omega_2 \,|\, \varepsilon_3 \,\cup\, \omega_3 \,|\, \varepsilon_2) \,|\, \varepsilon_1 \cdot \varphi_1 \cup \varphi_2 \cup \varphi_3) & \\
= \varepsilon_1 \sqcup (\omega_2 \,|\, \varepsilon_3 \,\cup\, \omega_3 \,|\, \varepsilon_2 \cdot \varphi_2 \cup \varphi_3) & (\text{def}) \\
= \varepsilon_1 \sqcup (\varepsilon_2 \sqcup \varepsilon_3) & (\text{def})
\end{array}
$$

$\square$